A layered architecture for flexible Web service invocation

Valeria De Antonellis¹, Michele Melchiori¹, Luca De Santis², Massimo Mecella², Enrico Mussi³,*†, Barbara Pernici³ and Pierluigi Plebani³

¹Università di Brescia, Via Brianza 38, 25123 Brescia, Italy
²Università di Roma ‘La Sapienza’, Via Salaria 113, 00198 Roma, Italy
³Politecnico di Milano, Piazza Leonardo da Vinci 32, 20133 Milano, Italy

SUMMARY

Web service composition is emerging as an interesting approach to integrate business applications and create intra-organizational business processes. Single Web services are combined to create a complex Web service that will realize the process business logic. Once the process is created, it is executed by an orchestration engine that invokes individual Web services in the correct order. However, Web services composing the workflow sometimes become unavailable during the run-time phase, blocking process execution. This paper describes an architecture that allows the flexible orchestration of business processes. With this approach, Web services composing the process can be automatically substituted with other compatible Web services during process execution. A methodology is defined to evaluate Web service compatibility based on interface matching, in order to select substitutable Web services. Copyright © 2005 John Wiley & Sons, Ltd.

KEY WORDS: Web services; orchestration; substitution; dynamic process evolution

1. INTRODUCTION

Web services are emerging as a key technology for the integration of applications within and across enterprise boundaries [1]. A Web service can be defined as a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service using SOAP messages as prescribed by the service’s description; such messages are typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards [2].

*Correspondence to: Enrico Mussi, Dipartimento di Elettronica e Informazione, Politecnico di Milano, Piazza Leonardo da Vinci 32, 20133 Milano, Italy.
†E-mail: mussi@elet.polimi.it

Contract/grant sponsor: MIUR

Copyright © 2005 John Wiley & Sons, Ltd.
With Web services it is possible to wrap and reuse legacy applications in inter-organizational processes and thus deploy services that can be easily used to create cooperative processes, i.e. a set of collaborating activities in order to reach a common goal. In this way Web services may be deployed by different providers and the overall process can be distributed across multiple organizations.

Many efforts have been made by the Web service community to define standards and protocols in order to enable system integration. Among others, emergent technologies such as BPEL4WS [3], WSCI [4] and BPML [5] enable Web service orchestration and choreography. In fact, these technologies make it possible to define abstract business processes and then execute their instances [6]. Unfortunately, none of these approaches allows Web service substitution at run time. The most interesting case is BPEL4WS‡, where the process designer can define an abstract process model composed of the typical workflow structures in which activities are specified according to the WSDL of the desired Web services. Once this model is created for each specified Web service, real deployed Web services must be selected to build an executable process.

However, this approach is unable to provide the basis for a real abstract process specification for two main reasons. First, given a Web service description required by the abstract process model, it is almost impossible to find more than one available Web service with exactly the same syntax: in fact, the process designer usually defines the abstract process model knowing a priori the set of Web services that will be used in the executable process. Second, at run time the executable process is fixed and there is no way to modify any invoked Web service.

Consequently, the differences between the desired and the real situation must be exploited by using abstract service definitions and then creating an architecture to dynamically invoke the concrete deployed services in place of the abstract ones. The idea is that each abstract service can be substituted with a compatible concrete service and then, during the process execution phase, the architecture chooses which concrete service to invoke.

In previous work [7–10] we developed theories and methodologies for dynamic Web service substitution and orchestration; the aim of this paper is to present the definition, design and implementation of the architecture for the dynamic management of cooperative processes, which we have developed in the Italian research project VISPO (Virtual-district Internet-based Service Platform). In this project, different organizations of a district interact on the basis of a cooperative process, where all activities are implemented using the Web service technology. Our architecture (and the implemented platform) (i) supports the definition of abstract processes and executes process instances, and (ii) allows automatic run-time Web service substitution. With our approach there is no a priori defined limit on the number of invocable Web services.

The prominent aspects of our approach are as follows.

- **Abstract service definition.** This allows us to categorize Web services and enables us to describe the cooperative process in an abstract way.

---

‡BPEL4WS is now under the control of the OASIS consortium and has been renamed as WS-BPEL. Even if this move has introduced some modifications to the specification, the conceptual model remains the same and such changes do not affect our work.
- **Concrete service** definition. Concrete services represent deployed Web services. They are organized according to the functionalities described by abstract services and realize the activities of the cooperative process.
- **Service compatibility** definition. At design time, a cooperative process consists of abstract services. During execution, these are substituted by concrete services that can be invoked. Substitution is only possible between compatible services, so a service compatibility evaluation methodology is necessary.
- **Orchestration** approach. In our system, the idea is that one organization decides to create an added-value service using existing Web services deployed in the virtual district by other organizations. This paper adopts an orchestration approach, which enables organizations to define and execute cooperative processes; this is preferable to a choreography approach, where organizations agree on the observable behavior of the process by defining the exchanged messages and the ordering constraints under which messages are exchanged, because there is no a peer-to-peer communication between organizations in the virtual district.

The rest of the paper is organized as follows. Section 2 introduces the case study used to validate our architecture. Section 3 describes how service compatibility is evaluated and presents our architecture for dynamic service substitution. Section 4 describes how a cooperative process is executed in our approach. Section 5 analyzes the performance of the deployed system. Finally, Section 6 draws some concluding remarks.

## 2. CASE STUDY

The case study adopted in this paper to describe a sample application of our methodology is the execution of a cooperative process in a virtual district, i.e. a productive district where the companies involved use Information and Communication Technology (ICT) to cooperate for business purposes. In particular, we consider a cooperative process called ‘**Advanced Purchasing**’.

The execution environment is an Application Service Provider (ASP) platform used by the virtual district to manage its business. It is supposed that similar ASP platforms are used in different districts. The ‘Advanced Purchasing’ process consists of three services coordinated to accomplish a common goal: automated, sophisticated processing of district purchase orders. Figure 1 shows the workflow defining the cooperative process used to test our methodology. The ‘Advanced Purchasing’ process can be viewed as a service in itself.

Each service in the process performs specific operations, while the district ASP coordinates their communication and synchronization, thus acting as a service orchestrator.

The three services comprising the cooperative process are described below. They provide all the operations necessary to implement advanced purchasing functionalities and are invoked by the process during its execution.

- **MRO Service**: generates a company consumption model§. The model is created by analyzing previous purchase documents (e.g. invoices), and collecting a normalized description of ordered products. Given a purchase order and a consumption model, the MRO Service can state if a

---

§In specific economic jargon, MRO refers to all consumable goods used by a company, e.g. pens, papers, gloves, hardware, etc.
product order belongs to the model and, in this case, returns the normalized product description which is needed by the other involved services. This service provides operations to perform the MRO generation and MRO validation activities.

- **e-Selling Service**: acts like a virtual catalog. Companies use the e-Selling Service to publish their catalogs or to search and buy products from other companies in the virtual district. Buyers can access the catalog, search for products and buy them directly through this service. Operations provided by this module enable the Search product and Buy directly activities.
• **Group Purchase Service**: allows the grouping of purchase orders from the various enterprises, in order to simplify and optimize purchase procedures. Furthermore, by using this service, enterprises increase their contractual power over their suppliers and obtain better discounts. This service enables the *Buy with group purchase* activity.

Although the *e-Selling* and *Group Purchase* services are strictly related to a particular ASP platform, *MRO* can be considered as a more general service. It is supposed that the ‘Advanced Purchasing’ process executed in a virtual district $\mathcal{A}$ can also be performed with an *MRO* service taken from another district $\mathcal{B}$.

As an example (used throughout the paper), in a specific ‘Advanced Purchasing’ process instance, deployed as an orchestration of Web services, the following interactions can take place:

1. the cooperative process starts with a purchase request;
2. the orchestrator invokes the *MRO* Web service to generate the correct consumption model;
3. the consumption model is generated and returned to the orchestrator;
4. the order is validated with respect to the consumption model and the orchestrator asks the *e-Selling* Web service if the purchase request can be satisfied;
5. the *e-Selling* Web service responds that products are not available;
6. the orchestrator invokes the *Group Purchase* Web service;
7. the *Group Purchase* Web service accepts the purchase order, executes it and returns the response to the orchestrator, thus ending the process.

Figure 2 shows messages exchanged between the orchestrator and Web services; in this case the orchestrator executes the cooperative process specification by sending messages to relevant Web services and correctly managing the responses.

It is also supposed that the *MRO* Web service becomes unavailable during a process instance execution. When this occurs, the orchestrator must substitute the original Web service with a new compatible one to carry on the process. The new service may have a different interface but, to be an effective substitute, it should implement the same functionalities. Figure 3 depicts the substitution scenario, in which the orchestrator retrieves the compatible Web service *MRO Utility* taken from district $\mathcal{B}$ and invokes it to proceed with the process execution.

The substitution is possible only if the *MRO* Web service has not yet started its execution; in our current approach, we do not take into account issues of roll-backing services that have started execution and fail in the middle, but we assume that for a stateful Web service substitution is only possible before initiating an exchange of messages with it (and therefore before starting to modify its state).

### 3. DESIGN ISSUES AND ARCHITECTURE

The main purpose of the proposed VISPO architecture is to implement a computational model based on the Service Oriented Computing [11] paradigm to execute a *cooperative process* with flexible and dynamic Web service invocation.

A cooperative process can be defined as a set of services which collaborate to reach a common goal. Each service executes a particular task and is invoked by the process execution engine. Invoking Web services flexibly means that the execution engine can replace them during process execution.
In this section we describe our approach to evaluating service compatibility and present the VISPO system architecture, which implements the compatibility evaluation and process execution with dynamic Web service substitution.

Three kinds of users interact with the VISPO system:

- users, who interact with our platform to execute cooperative processes;
- domain experts, who supervise the VISPO system to control and manage the performed activities;
- service providers, who create abstract and concrete services and publish them in the VISPO system registry.

### 3.1. Flexible service invocation

Flexible execution of an abstract cooperative process enables one or more Web services used in the process to be substituted when this is required or appropriate; for instance, this may be useful when:

- an updated version of an existing service is made available;
- a new service providing better features and compatible with a service already used in the process is published;
• a service used in the process is no longer available;
• at each execution of the cooperative process the most suitable service to implement an activity is selected from a set of equivalent services on the basis of criteria such as cost, availability, etc.

Obviously, a substitution should not involve any possible service, but only those with similar functionalities. We therefore introduce the concept of *compatibility class*, a set of services that can substitute one another in the execution of a process activity.

The substitution is possible when the to-be-substituted Web service has not yet started its execution; in our current approach, we do not take into account issues of roll-backing services that have started executing and are substituted in the middle, but we assume that for a stateful Web service substitution is only possible before initiating exchange of messages with it (and therefore before starting to modify its state).

For example, in Figure 3, the *MRO* service is substituted by the *MRO Utility* service, which has similar functionalities: i.e. the services belong to the same compatibility class.

We also define an approach to evaluate service compatibility and a mechanism to substitute Web services during their invocation.
3.1.1. Service description

The compatibility evaluation takes into account descriptions expressed through a suitable service description language.

WSDL is currently the de facto standard to describe Web service functionalities. A WSDL specification represents the structure of a Web service in terms of the operations provided and the data requested and returned for each of them. A WSDL portType defines a collection of abstract operations supported by the service, separating them from their actual implementation. In our approach, we suppose that services are described as WSDL 1.1 [12] documents and that each Web service has only one associated portType describing the operations implemented by the service. This means that a service presenting several interfaces (i.e. portTypes) must be represented as a set of Web services.

WSDL enables Web service publishers to separate the abstract definition of service functionalities from the specific details of implementation, such as service location and service access protocols [13]. We adopt the definition of abstract service for services restricted to the service-only interface, while fully described services are called concrete services.

This distinction is very useful in defining the cooperative process, as it allows the process to be described in terms of abstract services, and enables the VISPO run-time environment to decide which concrete services to invoke.

Our definition of abstract services also implies that they are composed of types, message and portType WSDL 1.1 elements. This is in agreement with the definition given in the BPEL4WS and BPML specifications and allows easy integration of abstract services into these orchestration languages.

In our system, an abstract service definition introduces an associated compatibility class whose members are concrete services. These services may either share the abstract service interface associated with the compatibility class, and therefore implement the abstract service, or have a different interface, to be used through a wrapper component.

A compatibility class is therefore created when an abstract service is registered; a concrete service is registered by associating it with one or more compatibility classes. The next subsection introduces the evaluation of the degree of compatibility among the members of a compatibility class and the representative abstract service.

3.1.2. Service compatibility

In VISPO, service compatibility is evaluated on the basis of semantic information related to service interfaces. For this purpose, each Web service is represented in terms of a descriptor extracted from its WSDL specification (see Table 1). Descriptors give a summary abstract view of services in terms of deployed operations and exchanged information. Approaches based on the use of descriptors are widely studied in the field of reusable software components [14] to find components in a library matching given requirements. A descriptor is formally defined by a name of service and a set of triplets:

\[
<\text{operation}(\text{OP}), \ \text{input entities}(\text{IN}), \ \text{output entities}(\text{OUT})> \]

where \( \text{OP} \) is the set of operations a service can perform, \( \text{IN} \) is the set of the input information entities, and \( \text{OUT} \) is the set of the output information entities.
Table I. WSDL entities and corresponding Descriptor concepts.

<table>
<thead>
<tr>
<th>WSDL entity</th>
<th>Descriptor concept</th>
</tr>
</thead>
<tbody>
<tr>
<td>PortType name</td>
<td>Descriptor name</td>
</tr>
<tr>
<td>Operation name</td>
<td>Operation name</td>
</tr>
<tr>
<td>Input message part name</td>
<td>Input entity name</td>
</tr>
<tr>
<td>Output message part name</td>
<td>Output entity name</td>
</tr>
</tbody>
</table>

```xml
<types>
  <element name="Category">
    <complexType>
      <sequence>
        <element name="categoryName" type="xsd:string"/>
        <element name="categoryID" type="xsd:decimal"/>
      </sequence>
    </complexType>
  </element>
</types>

<wsdl:message name="generateMroRequest">
  <wsdl:part name="businessName" type="xsd:string"/>
  <wsdl:part name="fileInput" type="xsd:string"/>
</wsdl:message>

<wsdl:message name="generateMroResponse">
  <wsdl:part name="mroID" type="xsd:string"/>
  <wsdl:part name="businessName" type="xsd:string"/>
  <wsdl:part name="productCategory" type="Category"/>
</wsdl:message>

<wsdl:portType name="Mro">
  <wsdl:operation name="generatorMro" parameterOrder="businessName fileInput">
    <wsdl:input name="generateMroRequest" message="impl:generateMroRequest"/>
    <wsdl:output name="mroGenerateResponse" message="impl.mroGenerateResponse"/>
  </wsdl:operation>
</wsdl:portType>
```

Figure 4. WSDL portType example.
Figure 4 shows an example of a WSDL portType definition, while Figure 5 shows the corresponding descriptor.

Descriptors are analyzed to compute similarity coefficients with values in the range [0, 1] so that the higher the similarity coefficient value, the higher the similarity of the services involved in terms of the operations they provide and the data they exchange with the invoking process.

The ARTEMIS system providing tools for semantic schema matching [15,16] is used to evaluate similarity. A similarity coefficient $GSim$ is calculated for a pair of service descriptors $S_i$ and $S_j$ as a weighted sum of an entity-based similarity coefficient $ESim(S_i, S_j)$ and a functionality-based similarity coefficient $FSim(S_i, S_j)$.
• **Entity-based similarity coefficient.** The Entity-based similarity coefficient of two descriptors $S_i$ and $S_j$, denoted by $ESim(S_i, S_j)$, is evaluated by comparing the input/output information they contain. Names of input and output entities are compared to evaluate their degree of affinity $A()$ (with $A() \in [0, 1]$). The affinity $A()$ between names is computed using a thesaurus of weighted terminological relationships (e.g. synonymy, hyperonymy) supported by ARTEMIS. Two entity names, $n$ and $n'$, have affinity if the thesaurus contains at least one terminological relationship path between them and the strength is equal to or greater than a given threshold. The higher the number of pairs of entities from the two services, with affinity, the higher their $ESim$ value.

• **Functionality-based similarity coefficient.** The functionality-based similarity coefficient of two descriptors $S_i$ and $S_j$, denoted by $FSim(S_i, S_j)$, is evaluated by comparing the operations contained in them. Here too, the comparison is based on the affinity $A()$ function. Two operations have affinity if their names, input information entities and output information entities have affinity in the thesaurus. Their affinity value is evaluated on the basis of the affinity values of their corresponding descriptor elements. The higher the number of pairs of operations from the two services, with affinity, the higher their $FSim$ value.

As an example of operation similarity evaluation, the reader should consider the operations

```plaintext
generateMro
  input={businessName, fileInput}
  output={mroID, businessName}

createMro
  input={commercialName, inputFilePath}
  output={mroID, commercialName}
```

According to our analysis, these two operations are similar and have affinity 1 (see Table III), as:
(i) $A() = 1$ for pairs of names that are synonyms; (ii) the operation names are synonyms; (iii) for each I/O entity of `generateMro` there is a corresponding synonym entity in `createMro` and vice versa. A weaker correspondence between the names and the I/O entities of two operations would be given a lower affinity value (as for `validateMro` and `checkMro` in Table III).

Note that a high similarity value between service descriptors does not guarantee full automatic substitutability between the corresponding services. In fact, there could be syntactic differences among the interfaces involved, i.e. different operation or parameter names, number or positions (as shown in Figure 3, where the MRO Utility and MRO services differ in the number and name of operations). Therefore, besides the descriptor similarity evaluation, mapping information must be created to take account of the differences between the abstract and concrete service interfaces. Figure 6 gives an example of mapping information.

It is worth noting that both evaluation of service compatibility and generation of mapping information follow a 1:1 approach. Given two services, each element of the first is compared with
each element of the second and the relative mapping information is generated; each operation is thus compared with a single operation, not with the composition of two or more other operations.

3.1.3. Service registration and retrieval

A Web service is published by registering its WSDL specification in the VISPO Registry and specifying its compatibility classes. Different steps are followed according to the type of service the provider wishes to publish. Specifically, the publication of an abstract service involves the definition of a new associated compatibility class; in contrast, service providers wishing to publish a concrete service must first define their Web service interface and then register it in one or more existing compatibility classes.

Besides the usual publication mechanisms, the VISPO Registry generates the correspondent descriptor for each registered service and performs its similarity evaluation on abstract services corresponding to the compatibility classes in which the service is inserted. The descriptor generation phase is transparent to the publisher, to maintain the compatibility evaluation logic within the system. For each compatibility class in which the concrete service is inserted, similarity values are computed and mapping information generated.

3.1.4. Wrappers and service invocation

Once compatibility has been evaluated and mapping information generated, concrete services can be invoked. Our invocation technology allows each abstract service interface to be mapped to a concrete
service implementation and enables the cooperative process run-time environment to act as if it were invoking the abstract services.

When a running cooperative process activity has to be executed and the corresponding abstract service is invoked, two different cases are possible:

- the concrete service interface is the same as the abstract one, therefore the system can locate and invoke the concrete service directly;
- the abstract and concrete interfaces are different, therefore a wrapper is needed to map abstract operations/parameters to concrete operations/parameters.

An interesting case is when a concrete service has a similarity coefficient value equal to 1. In this case, even if the concrete service fully satisfies the requirements of the abstract service, a wrapper may be necessary. In fact, the similarity coefficients do not consider the order of the operation parameters, and a maximum value may be given to a concrete service even if the concrete operation signatures do not completely match the abstract operation signatures. In this case, a wrapper solving these discrepancies is required.

Domain experts have to create wrappers on the basis of the mapping information. A facility in the VISPO system allows creation of a wrapper skeleton that the domain experts can check/modify to obtain the desired adaptation result (see Section 3.3).

3.2. Architecture

In this section, we introduce the VISPO architecture that supports flexible Web service invocation.

Figure 7 shows the system architecture, which consists of five main components:

- the VISPO Registry to register and search Web services and their descriptions;
- the Compatible Service Provider (CSP) to retrieve all concrete services belonging to a compatibility class with the mapping information needed for service substitution;
- the Compatibility Module to evaluate descriptor similarity and create mapping information;
- the Invoker for flexible invocation of concrete services;
- the Orchestration Engine to orchestrate service invocation.

In summary, the activities in the process are executed on the basis of the Invoker’s functionalities: it receives an abstract service definition as input and uses the CSP to find all compatible services and related mapping information. A concrete service is then selected and invoked.

Presented below are details of the VISPO Registry, the CSP and the Invoker, while the Orchestration Engine is described in Section 4.

3.2.1. VISPO Registry

The VISPO Registry can be used to publish and retrieve Web service specifications, group services into compatibility classes and manage service descriptors.

As Figure 8 shows, the VISPO Registry consists of a database, called descriptor base, and a UDDI v2 Registry [17]. Both parts are accessed through the VISPO Registry API, which extends the methods of UDDI v2 API [18] to support descriptor management and maintain compatibility with the UDDI v2
API specification. In this way, the registry can still be used by a UDDI v2 standard client to publish and retrieve service information, while only extended clients can access descriptors using registry specific methods.

We decided to use the UDDI v2 Registry because the UDDI v3 [19] Registry specification was still under development (at the time of this work) and client libraries for coding the VISPO Registry API were not available. However, our choice does not preclude use of a UDDI v3 Registry to realize the UDDI component of the VISPO Registry.

Service publishing is one of the most important operations, and differs slightly for abstract and concrete services. However, both are categorized using the features provided by UDDI v2 registries. Using the categoryBag and its subelement keyedReference correctly [20], enables a general keyword taxonomy to be defined and the published services to be categorized.

The keyedReference structure consists of three elements:

- \textit{tModelKey}, which specifies the \textit{tModel} that defines the taxonomy used for the categorization; we use the \textit{UUID:A035A07C-F362-44DD-8F95-E2B134BF43B4} value, which implies the use of a general keyword taxonomy;
- \textit{keyName}, which specifies the name of the general keyword taxonomy to be used for service categorization; we use the \textit{vispo-category:types} value that states the VISPO taxonomy use;
• keyValue, which specifies the category value; this represents the name of the compatibility class in which the published service is inserted.

The same approach can also be used even with an ebXML Registry [21], one of the most important alternatives to UDDI. In fact, ebXML Registry Model includes the option, through the so-called classification scheme, of introducing a specific way to organize the information stored in such a registry. Guidelines for abstract and concrete service publication are described below.

Abstract services can be published by saving a tModel referring to the WSDL abstract description of the service in the VISPO Registry. The tModel is characterized by a categoryBag with a keyedReference in which the keyValue element contains the name of the compatibility class defined by the published abstract service.

Once the abstract service is published, the descriptor is automatically created while the VISPO Registry checks that every new abstract service defines a new compatibility class. A compatibility class must not have more than one abstract service. The registry analyzes the published contents and notifies the abstract service publisher if inconsistencies are detected.

Concrete services publication requires registration of a tModel, which refers to the complete WSDL description including the binding information and specifies the compatibility classes in which the concrete service is inserted. The tModel is characterized by a categoryBag with a series of keyedReference, each defining a compatibility class to which the concrete service belongs. The name of the compatibility class is inserted into the keyValue element and the VISPO Registry checks that each defined keyValue refers to an existing compatibility class. Otherwise the registry notifies the concrete service publisher.
When a concrete service is published, descriptor and mapping information is generated. While the descriptor is generated by the registry itself, mapping information is generated by the CSP. Given a published concrete service, the VISPO Registry passes it to the CSP, which evaluates its similarity to the abstract services of the compatibility classes in which it was inserted and generates the mapping information. The CSP is described in depth in the next section. However, to avoid incorrect publications and create homogeneous compatibility classes, the domain expert must check the submitted keyValue values and generated information to verify that the concrete service is really suitable for inclusion in the specified compatibility classes.

3.2.2. CSP

The CSP is able to evaluate the compatibility of a given set of services with respect to a reference service and generate the corresponding mapping information. This is then saved in a dedicated registry and can be accessed by the CSP to retrieve useful service compatibility information.

In the VISPO system, the CSP is used during the publication and invocation phases. In the publication phase, it is used by the VISPO Registry to evaluate the similarity of the published concrete service to the abstract services defining the compatibility classes in which the service is published. In the invocation phase, the Invoker uses the CSP to retrieve the similarity values of concrete services belonging to the same compatibility class, in order to rank and select which concrete service to invoke.

The CSP architecture, depicted in Figure 9, consists of four main components.

- **Service Research Module.** The main function is to serve as a proxy for the Compatibility Engine with respect to the VISPO Registry. This module provides various methods to browse the registry and retrieve service information useful for the Compatibility Engine. It uses the simple inquiry API of the VISPO Registry to create more complex operations to retrieve all the necessary service information in just one step. With this module it is possible:
  - given a service key, to retrieve its WSDL and its descriptor;
  - given a compatibility class, to retrieve all WSDL documents of services belonging to it;
  - given a compatibility class, to retrieve all descriptors of services belonging to it.

The registry, which can reside on a different host, is communicated with via HTTP, and all exchanged information is in XML format.

- **Compatibility Evaluation Module.** Supports the Compatibility Engine during service compatibility evaluation. It provides a set of optimized operations to facilitate tasks such as affinity evaluation and service filtering. Through these operations, which are built on top of the semantic descriptor evaluation and schema matching functionalities provided by the ARTEMIS [15,16] tool, the Compatibility Engine evaluates compatibility between services and generates mapping information.

The Compatibility Evaluation Module enables evaluation of the similarity of abstract and concrete descriptors (in terms of affinity between their operations, input/output parameters and data types).

- **CSP SOAP Interface.** As shown in Figure 9, the CSP can be accessed through a SOAP interface, allowing the CSP to be used as a Web service. This interface is built using a servlet container.

- **Compatibility Engine.** This is the core of the CSP. It contains the logic for searching concrete services and evaluating their affinity with the abstract service belonging to the same
compatibility class. The result is a list of compatible services with relating mapping information. The evaluation process consists of the following steps.

1. **Service research**: once the tModel key and compatibility class of an abstract service are received as input, the Compatibility Engine uses the Research Module to retrieve all the descriptors and WSDL descriptions of the concrete services belonging to the same compatibility class from the VISPO Registry.

2. **Descriptor matching**: using the Compatibility Evaluation module, the abstract service descriptor is matched against all concrete service descriptors and a similarity coefficient
Table II. Descriptor matching table related to the MRO compatibility class.

<table>
<thead>
<tr>
<th>Concrete descriptor</th>
<th>GSim</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mro</td>
<td>1.0</td>
</tr>
<tr>
<td>MroUtility</td>
<td>0.791</td>
</tr>
</tbody>
</table>

Table III. Operation matching table related to the MRO compatibility class.

<table>
<thead>
<tr>
<th>Abstract operation</th>
<th>Concrete operation</th>
<th>Affinity</th>
<th>Selected</th>
</tr>
</thead>
<tbody>
<tr>
<td>generateMro</td>
<td>createMro</td>
<td>1.0</td>
<td>y</td>
</tr>
<tr>
<td>validateMro</td>
<td>checkMro</td>
<td>0.9</td>
<td>y</td>
</tr>
<tr>
<td>none</td>
<td>saveMro</td>
<td>0.0</td>
<td>n</td>
</tr>
</tbody>
</table>

$GSim$ calculated for each match (see Table II). The higher the $GSim$ value, the higher the probability that the concrete service will substitute the abstract one. Only services with $GSim$ greater than a threshold $t_c$ are selected for the next steps, in which the mapping information is generated. This and all other thresholds used in the Compatibility Evaluation Module are decided by experimentation and are proposed to the VISPO platform user, who can modify them in any case to improve control of the compatibility evaluation process.

3. For each selected concrete service the following (sub)steps are performed.

(a) Operation matching: in this step, for each selected concrete service and operation required by the abstract service, the engine selects the concrete service operation with the highest affinity value. Specifically, the Compatibility Engine selects only operations with an affinity value greater than a given threshold $t_{op}$. As illustrated in Table III, the number of operations in the concrete and abstract service may not be equal. For instance, a concrete compatible service provides an operation not required by the abstract service, or a concrete service provides fewer operations than required by the abstract service. Whereas in the former case the concrete service can be selected even if there is an operation that will never be invoked, in the latter the concrete service must be discarded, because we assume that the minimal substitutable part is the service, not the operation.

(b) Input parameter matching: for each operation in the abstract service, the Compatibility Engine creates an affinity table in which all input parameters are compared with those of a selected concrete operation. The relation between two parameters is considered to be valid if the affinity value is greater than the given threshold $t_{ip}$ (see Table IV). If the number of abstract parameters is greater than or equal to the number of selected concrete parameters, the operations can be
Table IV. Input parameter matching table related to the MRO compatibility class.

<table>
<thead>
<tr>
<th>Abstract parameter</th>
<th>Concrete parameter</th>
<th>Affinity</th>
<th>Selected</th>
</tr>
</thead>
<tbody>
<tr>
<td>businessName</td>
<td>name</td>
<td>0.8</td>
<td>y</td>
</tr>
<tr>
<td>businessName</td>
<td>input</td>
<td>0</td>
<td>n</td>
</tr>
<tr>
<td>businessName</td>
<td>category</td>
<td>0</td>
<td>n</td>
</tr>
<tr>
<td>inputData</td>
<td>name</td>
<td>0</td>
<td>n</td>
</tr>
<tr>
<td>inputData</td>
<td>input</td>
<td>1</td>
<td>y</td>
</tr>
<tr>
<td>inputData</td>
<td>category</td>
<td>0</td>
<td>n</td>
</tr>
<tr>
<td>productCategory</td>
<td>name</td>
<td>0</td>
<td>n</td>
</tr>
<tr>
<td>productCategory</td>
<td>input</td>
<td>0</td>
<td>n</td>
</tr>
<tr>
<td>productCategory</td>
<td>category</td>
<td>0.8</td>
<td>y</td>
</tr>
</tbody>
</table>

substituted automatically. Otherwise, the operation can still be used, but in the runtime phase, the system will have to ask the cooperative process user to specify the input values for the extra parameters. In both cases the operation is accepted and the Compatibility Engine proceeds to the output parameter matching step.

(c) **Output parameter matching:** the Compatibility Engine compares the output parameters of each abstract operation with those of the selected concrete operation, and for every comparison an affinity table is created. The relation between two parameters is considered valid if the affinity value is greater than the given threshold $t_{op}$. If the number of abstract parameters is less than or equal to the number of selected concrete parameters, the operations can be automatically substituted. Otherwise, the concrete operation cannot be used for service substitutability. In this case the concrete service must be rejected.

(d) **Data type matching:** this is the last step of the evaluation process. The Compatibility Engine analyzes the structure and evaluates the affinity of the pair of parameters identified in the previous step. Affinity is evaluated only if both parameters have the same structure (i.e. both simple or both complex), otherwise the concrete service is rejected. Complex structures are evaluated by calculating the affinity between the simple types which compose them. The greater the number of simple types with affinity, the higher the complex structure affinity. As in previous steps, relations between parameters are considered valid only if their affinity value is greater than or equal to a given threshold $t_{sp}$. The abstract service can be substituted by the analyzed concrete service only if all its parameters are related to those defined by the concrete specification.

The CSP is entirely written in Java and uses JAXB [22] technology to create a set of classes and interfaces to parse, access and manipulate XML documents. WSDL documents are first accessed using a JWSDL [23] compliant library and then translated into descriptors using JAXB. Descriptor classes are then accessed from the Compatibility Evaluation Module to evaluate compatibility. During this process, JAXB is used to generate and serialize mapping information.
3.2.3. Compatibility module

The main purpose of the Compatibility Module is to implement the evaluation of the semantic service similarity coefficients. As shown in Figure 9, it is invoked by the CSP, to which its evaluations are returned. These are used by the domain experts at the publication phase to check the correct assignment of service to compatibility classes. At the execution phase, semantic service similarities are also taken into account to retrieve the services in a compatibility class most suitable for executing a required process activity.

In our architecture this module is realized by wrapping the ARTEMIS system [15,16] provided with a CORBA interface. As shown in Figure 9, the Compatibility Module is invoked by the CSP using CORBA communication.

ARTEMIS is used to evaluate descriptor similarity and terminology affinities in general. The first functionality allows the CSP to compare service descriptors in order to evaluate semantic similarity coefficients for the analyzed services. The functionality of term affinity evaluation is used to compare the terms occurring within the service interfaces in order to construct mapping information between the abstract and concrete services that the CSP proposes to the domain expert as wrapper skeletons on which complete wrappers are built (see Section 3.3).

For the similarity and affinity computation, ARTEMIS exploits a thesaurus in which terminological relationships (such as synonym-of, broader-than, and related-to) are established among domain terms. In our trial the thesaurus is filled with domain knowledge provided by domain experts.

This module is not used directly by the Compatibility Engine, but is accessed by the Compatibility Evaluation Module which uses the functionality of term affinity evaluation to build more complex functions.

3.2.4. Invoker

Once the mapping information, the WSDL description of the abstract service, and a list of compatible services have been given, the Invoker’s main purpose is to manage concrete service invocation. As described in Section 3.1.4, the Invoker receives an abstract invocation request based upon an abstract service definition, selects a concrete service, invokes the service through its wrapper and returns the response as defined by the abstract interface definition.

The Invoker architecture, depicted in Figure 10, comprises four main modules.

- **Invoker HTTP Interface**: the methods exposed through this interface allow linking of an abstract service, invocation of an abstract service, unlinking from an abstract service, and retrying a previously linked abstract service (see Section 4).
- **Invoker Engine**: this is the core of the Invoker and contains the module’s application logic. Given an abstract service definition and a list of compatible concrete services, ordered by degree of similarity, it starts to invoke the service with the highest similarity value. If this invocation fails, it tries the second service in the list and so on. The invocation process is started by retrieving the wrapper designed for the selected concrete service and passing it to the Service Invoker module.
- **Service Invoker**: this module invokes concrete services. It is activated by the Invoker Engine and performs the invocation using the assigned wrapper.
A LAYERED ARCHITECTURE FOR FLEXIBLE WEB SERVICE INVOCATION

Figure 10. Invoker architecture.

- **Wrapper Repository**: this contains all the wrappers associated with the concrete services. Wrappers are created semi-automatically. Using the mapping information generated by the system, domain experts can build wrappers and save them in this repository. When the *invoke* operation is performed, the Invoker Engine retrieves the correct wrapper and plugs it into the Service Invoker.
3.3. Wrapper creation

As stated in Section 3.1.4, wrappers are created by domain experts from mapping information. Given an abstract service, a concrete compatible service and related mapping information, domain experts must implement the specific interface that defines wrapper methods.

Domain experts implement the method logic, which allows abstract service invocations to be translated into concrete service invocations. These methods, used by the Service Invoker to perform effective service invocation, allow the system to:

- locate concrete service access points;
- translate abstract operation inputs into concrete operation inputs;
- translate concrete operation outputs into abstract operation outputs.

In some cases a concrete operation may require more input parameters than the abstract one. This means that it is not possible to invoke this operation with an abstract parameter translation alone: instead all missing parameters must be retrieved by requesting them from the cooperative process executor. The domain expert must create wrappers able to retrieve the additional information needed. In our prototype implementation, we request the missing parameters from process executors via e-mail, redirecting them to an HTML page where required values can be inserted.

4. COOPERATIVE PROCESSES AND FLEXIBLE INVOCATION

The execution of a cooperative process requires sophisticated interactions among services. In the VISPO system this task is demanded of the Orchestration Engine. This module provides workflow functionalities that drive the Invoker to correctly manage the information exchanges between services.

The Orchestration Engine operates on the basis of the abstract interface definition of the involved services, and the instantiation between abstract services and concrete services is automatically managed by the Invoker. When the Orchestration Engine initiates the execution of a new workflow instance, it asks the Invoker to link (i.e. to associate with the current process instance) specific concrete service instances of the required abstract services, and invokes their operations during the workflow; if a service is unavailable, the Invoker automatically substitutes it with a compatible one, and the Orchestration Engine is shielded by such a substitution. When the Orchestration Engine has finished with a service, it explicitly unlinks it, to enable the Invoker to release its resources (unplug wrappers, etc.).

Orchestration Engine and Invoker communication is based on HTTP protocol, to avoid incompatibilities due to the different SOAP implementations tried out during the system’s development.

From a methodological point of view, a cooperative process is designed by a Petri Net-based formalism, as described in [9]. The use of such a precise formalism allows verification of correctness properties, such as the absence of deadlocks and the consistency of each service at the end of the cooperative process enactment. The Petri Net-based schema is then translated into an effective orchestration technology in order to enact process instances. Among the many languages that have been proposed for orchestration [25], we concentrated on BPEL4WS [3]; the Petri-Net is therefore translated into two BPEL4WS-compliant files, one each for the orchestration interface and implementation [26]. Such files are the process schemas to be enacted by the Orchestration Engine.
Figure 11 shows the internal architecture of the Orchestration Engine. It consists of two main components, the Workflow Engine and the HTTP Interface Layer.

**Workflow Engine.** This module interacts with the HTTP Interface Layer through SOAP messages\(^\ddagger\). To develop the workflow engine, we experimented with the most commonly used BPEL4WS-compliant engines, BPWS4J [27] and Collaxa BPEL Server [28], finally adopting the Collaxa BPEL Server 2.0.

In Figure 12 we show a fragment of the final BPEL4WS orchestration implementation of the cooperative process described in Section 2. As can be seen, the Workflow Engine always invokes the HTTP Interface Layer (which is itself a Web service) and this layer sends the Invoker (through HTTP calls) all the information needed to correctly invoke services, by using the service’s abstract interface only. The application interface between the Orchestration Engine and the Invoker consists of the following operations.

- **link(abstractService) return serviceIdentifier**: the Orchestration Engine requires the Invoker to establish a connection with the specified abstractService. The Invoker returns the serviceIdentifier to be used for service invocation.
- **invoke(serviceIdentifier, operation[p1,...pn]) return operationResult**: when the Orchestration Engine needs to invoke an operation with parameter p1,...pn, obtaining the operationResult.
- **unlink(serviceIdentifier)**: when the Orchestration Engine releases a service that will no longer be used in the process.
- **retry(serviceIdentifier)**: when the Orchestration Engine wants to test the availability of a service.

\(^\ddagger\)In BPEL4WS, an orchestration is itself a Web service, and therefore communicates through SOAP messages. As we decided that the architecture should be completely HTTP based, we needed to ‘wrap’ such a protocol.
<process name="VispoDemo"
    suppressJoinFailure="yes"
    targetNamespace="http://progettovispo.com"
    xmlns="http://schemas.xmlsoap.org/ws/2003/03/business-process/"
    xmlns:tns="http://progettovispo.com"
    ........

    <partnerLinks>
        .......
        <partnerLink name="HTTPInterfaceLayer"
            partnerLinkType="cdp:HTTPInterfaceLayer"
            partnerRole="HTTPInterfaceLayerService"/>
        .......
    </partnerLinks>

    <flow>
        <sequence>
            .......
            <assign>
                .......
                <copy>
                    <from expression="Invoke"/>
                    <to part="operazioneInvoker" variable="invocationGMC"/>
                </copy>
                .......
                <from part="ServiceName" variable="InvokerResponseS1"/>
                <to part="ServiceName" variable="invocationGMC"/>
                .......
            </assign>
            <invoke name="invocaGMC"
                partnerLink="HTTPInterfaceLayer"
                portType="cdp:HTTPInterfaceLayer"
                operation="invocation"
                inputVariable="invocationGMC"
                outputVariable="responseGMC"/>
            .......
        </sequence>
        .......
    </flow>
    .......
</process>

Figure 12. BPEL file example.
HTTP Interface Layer. This component interfaces the Workflow Engine with the Invoker module. It processes the SOAP messages received in order to send the correct instructions to the Invoker. The HTTP Interface Layer receives the Orchestration Engine directives and converts them to HTTP messages; it then converts the responses received in HTTP back to SOAP. It can be viewed as a wrapper service between the HTTP protocol used by the Invoker and the SOAP protocol used by the Workflow Engine.

5. PERFORMANCE EVALUATION

An architecture prototype has been developed in the context of the VISPO project and a demo of its functionalities is available online at http://cube-si.elet.polimi.it/vispo/index.html, where a cooperative process is offered and the substitution feature can be tested. Although our implementation is still a prototype and more work is needed to make the VISPO system stable and efficient, we have undertaken some performance tests, measuring Invoker and the CSP response times.

Evaluations were performed separately as the compatibility evaluation function provided by the CSP is used only during the concrete service publication phase, while the Invoker functionalities are used during the process execution phase. In this sense, we could say that the CSP does not have any direct impact on the performance of the VISPO system during execution of cooperative processes.

5.1. Test environment

To study the CSP and Invoker module performance, we designed a test environment to simulate use of our system. The test environment consisted of seven dedicated boxes connected with a 100 Mbit LAN. The VISPO system was installed on a bi-processor Intel Xeon 2.4 GHz, 1 GB RAM, Win2000 server and deployed using Tomcat 4.1.27 (200 threads), while each of the Web services used to compose the test cooperative processes was deployed on a different workstation (processor Intel PIII, 256 MB RAM, SuSE 9.0). To evaluate the performance of the VISPO platform independently of the service execution times, the Web service methods considered in the test do not perform any business logic but simply return a fixed result.

We used TestMaker [29] to code the test clients; TestMaker provides an environment for building software to test Web services and applications. We built two types of application to test the Invoker and CSP separately. Below we describe how the tests were performed and report some results.

5.2. Invoker performances

The main aim of the Invoker module is to manage the invocation of the abstract services comprising a cooperative process. Given an abstract service, an abstract invocation request, an ordered list of compatible concrete services, and a set of wrappers, the Invoker manages the dynamic abstract service invocation. We performed two kinds of tests, the first to correlate process availability with the Invoker response time, and the second to evaluate the average Invoker response time under heavy load conditions.
5.2.1. Process availability and invoker response time

To relate the process availability with the Invoker response time we used a simple test process. This included a sequence of five equivalent activities performed using Web services with the same interface, delivered by different service providers, and executed on different servers (i.e. Web service execution is performed independently). The process ends successfully only if all its activities are performed without failures. The simplicity of the process does not influence the evaluation of the Invoker module performance, which is responsible for the invocation of abstract services only and not for the orchestration of cooperative processes.

Obviously, our methodology to manage the execution of cooperative processes is strictly dependent on the availability of the involved services. If the Web service executions are independent of one another, and have the same failure rate, the availability \( Av \) of the test process can be calculated using the formula:

\[
Av = (1 - P(WSF))^n
\]

where \( P(WSF) \) is the probability of a concrete service failure and \( n \) is the number of invoked services. In our case \( n = 5 \). Let us suppose that all Web services used during the execution of the process are retrieved from the WWW and that their failure rate is 16% [30]. If there is only one concrete service for each abstract service, the service substitution between concrete services cannot be performed and the process availability value is \( Av = 0.4182\% \). Increasing the number of the concrete services that can substitute an abstract service will reduce the probability \( P(WSF) \), increasing the availability \( Av \) of the process. Figure 13 reports the relation between the process availability and the number of concrete services.
services available for each abstract service. This is evaluated using independent Web services, also with a failure rate value of 16%.

The Invoker module invokes concrete services using wrappers to transform abstract to concrete parameters and \textit{vice versa}. This means that the time spent by a wrapper in translating parameters may be relevant to the Invoker response time. Wrapper response time depends on the complexity of the wrapper itself: the more complex the transformations performed by the wrapper, the higher the response time.

To evaluate the Invoker response time, we conducted two tests using a client deployed on a single workstation (Intel PIII processor, 256 MB RAM, SuSE 9.0). Each test simulates the invocation of an abstract service, with invocation of one to six alternative services, where we have supposed that the difference between the abstract and concrete services lies only in the name of the invoked operation and the name and types of exchanged parameters. In the first test we used one operation with one parameter, while in the second we used one operation with 10 parameters.

Figures 13 and 14 highlight that process availability tends to one, while the response time increases linearly with the number of concrete services.

Obviously, the Invoker response time does not depend only on the response time of the wrappers. Two other factors have an impact on the its performance: the time spent in retrieving a wrapper from the repository and the time spent by the Invoker in managing the invocation request.

Given an abstract service invocation, the mean time spent by the Invoker to manage the request is 20 ms, while the mean time spent to retrieve a wrapper is 47 ms. It should be noted that a wrapper must be retrieved for every invoked concrete service.
5.2.2. Invoker performance under heavy load conditions

Invoker performance under different load conditions was tested according to the following system configuration: five clients execute an instance of the same cooperative process concurrently (see Figure 15), each being deployed on a different workstation (Intel PIII processor, 256 MB RAM, SuSE 9.0). As seen in the figure, the test process is a simple sequence of activities performed by different services. It should also be noted (i) that the average Invoker response time was evaluated without
substituting concrete services, and (ii) that in running the test the Invoker had to retrieve and use different kinds of wrappers.

To evaluate the average Invoker response time under increasing load conditions, we initially performed a cycle of 10 cooperative process executions on the first client. The test was then repeated by running two clients concurrently, with each client performing a cycle of 10 cooperative process executions. This was repeated with up to five clients running concurrently. Figure 16 reports the experimental results.

The entire test was repeated using cycles of 100 process executions to evaluate the scalability of the Invoker module. The results are reported in Figure 17.

Figure 18 shows the response times for 10 and 100 execution cycles, normalized so that the reported times actually refer to a single cycle. This enables the minimal difference in response times to be considered with respect to the different number of scheduled execution cycles.

5.3. CSP performances

As already discussed, the aim of the CSP is to compare Web services, evaluate their similarity and create mapping information to build service wrappers.

Given two or more Web services, mapping information is created. Their affinity is evaluated, and must be recalculated only if at least one interface of the Web services involved has changed.

The main factor influencing the CSP performance is the WSDL structure complexity in the Web services whose compatibility is being evaluated. For this reason, we conducted a test to evaluate the CSP response time considering different kinds of WSDL.
Figure 17. Invoker average response time measured on 100 cycles.

Figure 18. Average Invoker response time for one process execution measured on 10 and 100 cycles.
The test was conducted on pairs of Web services and using a client built with TestMaker and deployed on a single workstation (Intel PIII processor, 256 MB RAM, SuSE 9.0). It performed considering ever more complex WSDL descriptions, so that more and more differences between descriptors were introduced, with the aim of augmenting the computation load on the CSP and evaluating its performance and scalability. Each pair of WSDL documents contains the same number of operations and parameters. The operation and parameter names are different and all parameters are simple types.

To evaluate the impact of the number of operations and parameters on the CSP performance, we performed three kinds of tests. All tests evaluated the average CSP response time using WSDL pairs with one to five operations. In the first test, each operation had one parameter; in the second, each operation had five parameters, while in the third each operation had 10 parameters. Figure 19 reports the experimental results.

6. CONCLUDING REMARKS

Dynamic invocation of Web services is a very important task executing cooperative processes.

In this paper, we introduced an architecture for Web service compatibility based on interface analysis. The architecture allows very dynamic managing of cooperative processes allowing Web service substitution during the run-time phase.
The approach can be extended by also considering the behavior of Web services, that is functional constraints on the possible order of operation invocations a service can be engaged in (referred to as conversations). In such a way, two Web services are compatible not only when their interfaces match, but also when all the conversations supported by the to-be-substituted Web service are supported by the substitute one. Preliminary results have been presented in [24], and in the future we will extend our prototype in such a direction.

Moreover, we will explore how to consider non-functional aspects of Web services during the compatibility evaluation phase, including quality of service aspects. Other improvements will regard the ability of our approach to perform more sophisticated similarity evaluation by also considering the composition of two or more operations, overcoming the 1:1 service mapping behavior.

In addition, the prototype will be enhanced by adding new features such as the ability to perform asynchronous calls, support of Web services described using WSDL 2.0 [31] and a new VISPO Registry implementation using a UDDI v3 [19] registry.

Using Web services described with WSDL 2.0 does not influence our methodology directly, as service descriptions can be translated from WSDL 1.1 to WSDL 2.0 without information loss. We can use services described with WSDL 2.0 by just changing the parsing functions used to translate WSDL descriptions into service descriptors and invoking and deploying services with WSDL 2.0 tools, without changing our compatibility evaluation methodology.

The new VISPO Registry implementation has no impact on the other architecture modules, because the UDDI v3 specification explains how to build a UDDI v3 registry with multiple version support, allowing use of UDDI v2 clients to access UDDI V3 registries. As well as giving guidelines on how to implement a registry, the UDDI v3 specification also defines some constraints on data type composition, which must be followed by UDDI V2 clients in order to interact with UDDI V3 registries. These constraints are satisfied by our VISPO Registry API.

ACKNOWLEDGEMENT

Part of this work was supported by MIUR, through the ‘Fondo Strategico 2001’ project VISPO and the ‘FIRB 2001’ project MAIS.

REFERENCES


